El rol de Docker para ejecutar pruebas automatizadas como parte de la Integración Continua

Paula Beatriz Olmedo,
Fernanda Noel Pucheta Moyano

McAfee Argentina
Av. La Voz del Interior 7000, X5000 Córdoba, Argentina
{paulabeatriz_olmedo, fernanda_pucheta}@mcafee.com.ar

Resumen La Integración Continua es una práctica de la Ingeniería de Software cuyo uso se ha incrementado en los últimos años a la par de las Metodologías Ágiles. En los equipos de desarrollo, es un pilar fundamental para lograr la agilidad.
El objetivo de este trabajo es plantear una solución para acompañar al proceso de Integración Continua, desde el punto de vista de la prueba del código. Se utilizarán tecnologías como Docker y TeamCity, para desplegar los Frameworks necesarios que permitan ejecutar las pruebas del producto en desarrollo.

Keywords: Integración Continua, BDD, Testing Automatizado, Build Server, Docker, Cucumber, Gherkin, Ruby, RSpec

1. Introducción

La Integración Continua consiste en el proceso de integrar automáticamente y con frecuencia las distintas partes de un proyecto de software. De esta manera, la detección de errores se realiza de manera temprana [1], y además posibilita la entrega de una versión del producto en cualquier momento que sea necesario [2]. Se debe aclarar que por integración se entiende el proceso de la compilación y prueba del código en la medida en que se lo genere, como así también su posterior despliegue y puesta en ejecución [3].

Parte del proceso de la Integración Continua consiste en realizar pruebas sobre el código producido. Existen diferentes enfoques sobre los cuales trabajar, como por ejemplo TDD (Test Driven Development), DDD (Data Drive Development) o BDD (Behavior Driven Development), con el cual se trabajará en este caso. [4] El BDD (Desarrollo dirigido por comportamiento) consiste en escribir las pruebas en lenguaje no técnico o en lenguaje de negocio, de modo que todos los involucrados en el proceso de desarrollo del producto, ya sean técnicos (desarrolladores, testers, DevOps, etc) o no técnicos (Product Owners, Product Managers, vendedores, etc) puedan entender de qué se está hablando [5]. Sin embargo, a pesar del uso de lenguaje coloquial las expresiones se basan una

estructura similar a la lógica de Hoare [6], ya que se basan en definir una precondición para llegar a una post-condición para el software que se va a construir; ambas condiciones deben cumplirse para que pueda decirse que el mismo cumple con los objetivos. En este contexto, dicha lógica se ve plasmada en una notación que se denomina “Given-When-Then” [7] [8].

- Given (dado): se provee el contexto necesario mediante una adecuada configuración del sistema, para el escenario en el cual se realizarán las pruebas.
- When (cuando): se especifican las acciones que se realizan al iniciar las pruebas.
- Then (entonces): se expresa el resultado esperado de dicha prueba. Si el resultado obtenido coincide con el esperado, la prueba pasa. En caso contrario, falla.

```
Scenario: Carga de notas
    Given: Soy un profesor de un curso
    When: Ingreso a la página de la universidad
    And: Selecciono una nota del 1 al 10
    Then: Debo ver la nota cargada para dicho alumno
```

Figura 1. Ejemplo de notación Given-When-Then

Es importante destacar que el objetivo principal de este modelo de desarrollo es escribir dichas pruebas antes de escribir el código fuente, es decir, la implementación del sistema.

Para implementar BDD pueden utilizarse diversas herramientas. A continuación, se mencionarán algunas de ellas, ambas basadas en el lenguaje de programación Ruby.

Cucumber es una de las herramientas existentes que permite escribir el código correspondiente para ejecutar las pruebas necesarias, es decir, permite la implementación automática de las pruebas de software. Gherkin es un lenguaje que se implementa con este Framework, encargado de definir el comportamiento del software. Esto lo hace a través de la notación “Given-When-Then”mencionada anteriormente, y separa el código de la implementación de las pruebas, de la especificación de las mismas [9].

Por otro lado, se encuentra RSpec. Si bien posee el mismo enfoque que Cucumber, se diferencia del mismo debido en que no separa la especificación de la implementación de las pruebas, lo que dificulta la interpretación de las mismas a los involucrados en el negocio [10].

Existen herramientas para ejecutar las pruebas automatizadas sin la necesidad de realizar un despliegue local. Un ejemplo de esto es Docker. El mismo permite la creación de «contenedores» a fin de desplegar con sencillez las dependencias necesarias que una aplicación puede llegar a necesitar [11], ya sean paquetes de software, configuraciones o archivos ejecutables.

Un contenedor es una instancia de una imagen previamente construida, y consiste en un mecanismo de empaquetado lógico, que permite aislar aplicaciones del entorno en el cual corren en realidad. Operan a nivel de Sistema Operativo, es decir que a diferencia de las máquinas virtuales no virtualizan hardware, sino que comparten el hardware del entorno en que se está usando. Por esta razón, puede verse a los contenedores como una abstracción de la capa de aplicación [12].

Como parte de una tarea de un "Build Server", una imagen de Docker puede ser fácilmente instanciada, de manera que pueda ejecutar una acción (compilar código, correr pruebas, entre otras) de forma automática e independiente a los recursos de la máquina local en que se ejecute.

Actualmente existen diferentes herramientas de este tipo, como por ejemplo, Jenkins, Bamboo, GitLab CI o TeamCity [13]. En este trabajo se utilizará TeamCity, software de JetBrains. Es necesario aclarar que este tipo de recurso es una pieza fundamental en el proceso de Integración Continua.

2. Ejecución de pruebas automatizadas

El contexto de este trabajo se da dentro del desarrollo de un producto de software del tipo SIEM (Security Information and Event Management - Sistema de Gestión de Eventos e Información de Seguridad). [14]

El proceso de desarrollo del mismo utiliza distintos tipos de tecnologías, debido a su gran tamaño y complejidad, repercutiendo también en el proceso de realización de pruebas. Es por esto que se da una coexistencia de pruebas automatizadas escritas tanto en Cucumber como en RSpec.

Como se imaginará, al tratarse de Frameworks distintos, demandan instalaciones y configuraciones distintas. Esto implica gran cantidad de esfuerzo y conocimiento, lo cual torna el proceso tedioso y poco eficiente.

Es por ello que se pensó en una solución que incluyera la creación de dos imágenes de Docker, que desplegaran cada una lo necesario para el uso de dichos Frameworks. El despliegue de estas imágenes se da a partir de un archivo especial propio de Docker, denominado Dockerfile.

Se pensó en dos imágenes independientes, para mantener la modularidad del despliegue, aunque al estar ambos Frameworks basados en Ruby, comparten la incorporación de un archivo denominado Gemfile, el cual tiene la función de mantener las librerías necesarias en cada caso. Además, ambos Frameworks precisan ciertas librerías de bases de datos para su funcionamiento.

En la Figura 2 se detalla la conexión entre estos componentes, y cómo se integran al proceso de desarrollo del producto.

Se estima que la creación de estas imágenes no sólo facilitará el trabajo personal (al no necesitar conocimiento técnico para la configuración de los Frameworks), sino que también posibilitará la independencia de ciertos factores, como el hardware de cada máquina, o la conexión a Internet.

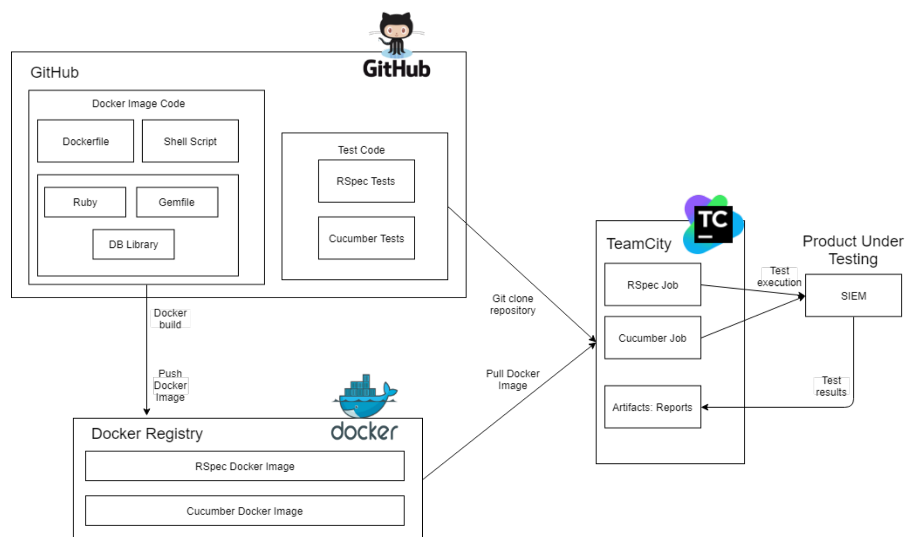

Figura 2. Arquitectura del trabajo realizado

2.1. Ejecución con RSpec

Además de incluir los archivos mencionados anteriormente (Gemfile y librerías de bases de datos) para la creación de la imagen de Docker correspondiente al Framework RSpec, también fue necesario agregar determinados archivos en formato JSON, que ayudan a la configuración del entorno sobre el cual dichas pruebas se ejecutarán. Partiendo entonces de estas premisas, se pensó la estructura del Dockerfile para que incluyera este requisito, además de los ya mencionados en el apartado anterior.

Se desarrolló también un script personalizado para utilizar como punto de entrada del Dockerfile [15] con el objetivo de ejecutar automáticamente las pruebas cada vez que el contenedor sea instanciado y poder así integrarlo en un Build Server. Además de la ejecución de pruebas, el script tiene la función de generar los reportes necesarios derivados de éstas pruebas.

2.2. Ejecución con Cucumber

Por su parte, la creación del archivo Dockerfile para Cucumber se pensó teniendo en cuenta los aspectos mencionados en el apartado 2. El mismo contiene como punto de entrada un script personalizado que permite ejecutar las pruebas. Además dicho script genera reportes que facilitan la visualización de los resultados a través de llamadas a los perfiles definidos en un archivo propio del Framework, denominado cucumber.yml

El archivo cucumber.yml posee los comandos más utilizados en un proyecto. Por convención, el mismo se encuentra ubicado en un subdirectorio denominado .config, dentro del directorio de trabajo actual.

Dentro del archivo cucumber.yml se definen perfiles. Los mismos poseen un nombre y una lista de los comandos que se desean ejecutar. Se explicará el uso de dichos perfiles, mediante el siguiente ejemplo:

```
#config/cucumber.yml
##YAML Template
html_report: --format pretty --format html --out=features_report.html
test: --tags @my_tests --tags "not @MANUAL" --tags "not @WIP"
```

Figura 3. Ejemplo de cucumber.yml

Cucumber posee complementos para mostrar los resultados de la ejecución de las pruebas en distintos formatos, como ser JSON o HTML. En este caso, el perfil «html_report» especifica que el formato de salida será un html.

Por su parte, el perfil «test» ejecutará todas las pruebas que estén rotuladas con la etiqueta «@my_tests», y no ejecutará aquellas que aún no estén automatizadas (rotuladas con la etiqueta “not @MANUAL”) o que se encuentren en proceso (@WIP, cuyas siglas significan “Work In Progress”) [16].

Una funcionalidad importante de la imagen de Docker de Cucumber, es que permite ingresar por parámetro las etiquetas de las pruebas que se deseen ejecutar.

2.3. Integración con TeamCity

El paso final para la conexión de Docker con TeamCity es la creación de una “tarea”, en dicho Build Server.

La configuración parte de un repositorio en Github que contiene la lógica de las pruebas [Version Control Settings]. El servidor permite elegir qué punto del código utilizar, es decir, la rama y los últimos cambios de la misma.

En segundo lugar se configuran “Build Steps”, los cuales en este caso se encargan de descargar el contenedor de Docker para luego ejecutar automáticamente las pruebas, según una serie de parámetros presentes en el script personalizado anteriormente mencionado.

Finalmente, un agregado en este tipo de ejecuciones es la creación de artefactos, los cuales consisten en archivos resultantes del uso del Build Server. En el caso de los Framworks RSpec y Cucumber, los artefactos que se construyen son reportes. Se pensó en la inclusión de distintos formatos (.json, .html, .xls, .xml) cada uno proveyendo distintos puntos de vista de la información generada.

Los reportes html generados son los más utilizados en este proyecto, ya que en ellos se puede observar de manera gráfica, el porcentaje de pruebas que pasaron y que fallaron, proveyendo más información de aquellas con resultado negativo, como por ejemplo, en qué parte del código fallaron y por qué.

3. Conclusión y trabajo futuro

Se concluye que aún es necesario realizar una investigación más profunda con respecto a aspectos tecnológicos de la integración continua. Como por ejemplo, ahondar más en el uso de Docker como creador de imágenes y contenedores así como también en el uso de Build Servers que permiten su integración, ya que se reconoce que las soluciones actualmente implementadas pueden optimizarse.

Un aspecto a optimizar, sería por ejemplo automatizar el proceso de creación de imágenes de Docker mediante, también, Teamcity. De modo que dicha creación sea el punto de entrada para la los jobs de ejecución de pruebas. Actualmente las imágenes son construidas de manera manual por quien lo necesite siendo necesario clonar el repositorio donde se encuentran los archivos que permiten este proceso, y tener un mínimo conocimiento para realizar dicha tarea. Automatizando el proceso, se prescindiría de todo eso.

No obstante esto, lo logrado hasta el momento facilita el trabajo de quienes necesitan realizar pruebas de las distintas partes del producto, ahorrándoles gran cantidad de tiempo y esfuerzo, que se utilizaría para la configuración de cada Framework en particular. Por otra parte el hecho de que las pruebas se ejecuten mediante un Build Server, torna el proceso más rápido y permite al usuario trabajar sin retrasos ya que la ejecución no consume recursos de la máquina local, y otorga la libertad de ejecutar un conjunto extenso de pruebas sin la necesidad de mantenerse conectado a la red durante dicho proceso (lo cual es indispensable cuando las pruebas son ejecutadas de manera local, donde si la persona se desplaza de su puesto de trabajo y pierde la conexión a Interet, se detiene el proceso.)

Finalmente, no debe dejar de mencionarse el favorable hecho de que los resultados de la ejecución de las pruebas (tanto en reportes finales como en tiempo

real a medida que las mismas se realizan) estén disponibles para cualquier miembro del equipo, contrastando así con la ejecución local, donde sólo el autor de la ejecución puede ver lo ocurrido.

Referencias

1. Martin Fowler, https://www.martinfowler.com/articles/continuousIntegration.html
2. Agile Alliance, https://www.agilealliance.org/glossary/continuous-integration
3. Thoughtworks, https://www.thoughtworks.com/continuous-integration
4. GaboEsquivel, https://gaboesquivel.com/blog/2014/differences-between-tdd-atdd-and-bdd/
5. Dan North, https://dannorth.net/introducing-bdd/
6. Wikipedia, https://en.wikipedia.org/wiki/Hoare_logic
7. Agile Alliance, https://www.agilealliance.org/glossary/bdd
8. Agile Alliance, https://www.agilealliance.org/glossary/gwt
9. GenBeta:Dev, https://www.genbetadev.com/metodologias-de-programacion/bdd-cucumber-y-gherkin-desarrollo-dirigido-por-comportamiento
10. Relish, https://relishapp.com/rspec
11. Docker Documentation, https://docs.docker.com/get-started/
12. Google Cloud, https://cloud.google.com/containers/
13. Vladimir Pecanac, https://code-maze.com/top-8-continuous-integration-tools
14. Wikipedia, https://en.wikipedia.org/wiki/Security_information_and_event_management
15. Docker Documentation, https://docs.docker.com/engine/reference/builder/#entrypoint
16. Cucumber Github, https://github.com/cucumber/cucumber/wiki/cucumber.yml